A Preliminary Evaluation of Cache-Miss-Initiated Prefetching Techniques in Scalable Multiprocessors

Ricardo Bianchini and Thomas J. LeBlanc

Technical Report 515
May 1994

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
May 1994

3. REPORT TYPE AND DATES COVERED
technical report

4. TITLE AND SUBTITLE
A Preliminary Evaluation of Cache-Miss-Initiated Prefetching Techniques in Scalable Multiprocessors

5. FUNDING NUMBERS
ONR N00014-92-J-1801
ARPA HPCC, Order 8930

6. AUTHOR(S)
Ricardo Bianchini and Thomas J. LeBlanc

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Computer Science Dept.
University of Rochester
734 Computer Studies Bldg.
Rochester, NY 14627-0226

8. PERFORMING ORGANIZATION REPORT NUMBER
TR 515

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research	ARPA
Information Systems	3701 N Fairfax Drive
Arlington, VA 22217	Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution of this document is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
(see title page)

Accession For	
NTIS GRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

14. SUBJECT TERMS
remote access latency; cache-coherent multiprocessors; spatial locality; stride-directed prefetching; software prefetching; execution-driven simulation

15. NUMBER OF PAGES
25 pages

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18

A Preliminary Evaluation of Cache-Miss-Initiated Prefetching Techniques in Scalable Multiprocessors

Ricardo Bianchini and Thomas J. LeBlanc

ricardo@cs.rochester.edu, leblanc@cs.rochester.edu

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 515

May 1994

Prefetching is an important technique for reducing the average latency of memory accesses in scalable cache-coherent multiprocessors. Aggressive prefetching can significantly reduce the number of cache misses, but may introduce bursty network and memory traffic, and increase data sharing and cache pollution. Given that we anticipate enormous increases in both network bandwidth and latency, we examine whether aggressive prefetching triggered by a miss (*cache-miss-initiated prefetching*) can substantially improve the running time of parallel programs.

Using execution-driven simulation of parallel programs on a scalable cache-coherent machine, we study the performance of three cache-miss-initiated prefetching techniques: large cache blocks, sequential prefetching, and hybrid prefetching. Large cache blocks (which fetch multiple words within a single block) and sequential prefetching (which fetches multiple consecutive blocks) are well-known prefetching strategies. Hybrid prefetching is a novel technique combining hardware and software support for stride-directed prefetching.

Our simulation results show that large cache blocks rarely provide significant performance improvements; the improvement in the miss rate is often too small (or nonexistent) to offset a corresponding increase in the miss penalty. Our results also show that sequential and hybrid prefetching perform better than prefetching via large cache blocks, and that hybrid prefetching performs at least as well as sequential prefetching. In fact, given sufficiently high bandwidth and regular memory addressing, hybrid prefetching can perform as well as software prefetching (which does not require a miss to initiate prefetching). We conclude that among the cache-miss-initiated prefetching techniques we consider, hybrid prefetching is the only technique that offers significant performance improvements for scalable multiprocessors.

This research was supported under ONR Contract No. N00014-92-J-1801 (in conjunction with the ARPA HPCC program, ARPA Order No. 8930) and NSF CISE Institutional Infrastructure Program Grant No. CDA-8822724. Ricardo Bianchini is supported by Brazilian CAPES and NUTES/UFRJ fellowships.

1 Introduction

The high cost of remote memory accesses is a major impediment to good performance on scalable cache-coherent multiprocessors. In order to tolerate the latency of remote memory accesses in these machines, data prefetching techniques triggered by cache misses are often used. We refer to these techniques as *cache-miss-initiated prefetching.*

Every prefetching technique is based on an ability to predict, in advance, which addresses an application will reference in the near future. If the predictions are wrong, then the cache is filled with data that will not be referenced soon, resulting in cache pollution. If data is prefetched too early, then it can become stale before it is referenced, requiring a refetch of the data and increasing coherence traffic. Prefetching techniques must balance the benefits of fetching data early with these increased costs.

Two additional characteristics of cache-miss-initiated prefetching techniques are: (1) some cache misses are *required*, since misses provide the only opportunities for prefetching and (2) a large amount of data must be transferred at each miss in order to prevent future misses. These two characteristics of cache-miss-initiated prefetching cause the data traffic of an application to become bursty, since there are fewer misses, but each miss prefetches lots of data. This bursty traffic can result in serious performance degradation, particularly in machines with limited communication or memory bandwidth. Thus, when considering aggressive cache-miss-initiated prefetching, there is an additional tradeoff between lower miss rates and the potential for network and memory contention.

In this paper we use execution-driven simulation of parallel programs to evaluate these tradeoffs for scalable multiprocessors with high network bandwidth and latency. In particular, we consider the effect on application performance of three different cache-miss-initiated prefetching techniques: (1) large cache blocks, which fetch multiple addresses within a single block, (2) sequential prefetching, which fetches multiple consecutive blocks, and (3) hybrid prefetching, a novel technique combining hardware and software support for stride-directed prefetching.

Our results show that block sizes between 16 and 128 bytes provide the best performance for our applications; larger blocks either increase the miss rate or incur an increase in the miss penalty that dominates any improvement in the miss rate. Our results also show that sequential and hybrid prefetching perform better than prefetching via large cache blocks, and that hybrid prefetching performs at least as well as sequential prefetching. In fact, hybrid prefetching can perform as well as software prefetching, given sufficient bandwidth and regular memory addressing. Based on these results, we conclude that among the cache-miss-initiated prefetching techniques we consider, hybrid prefetching is the only strategy that can offer significant performance improvements for scalable multiprocessors.

The remainder of this paper is organized as follows. In section 2 we describe in detail each of the cache-miss-initiated techniques we consider. In section 3 we describe our simulation methodology, performance metrics, and application workload. We present our experimental results in section 4, and our conclusions in section 5.

2 Cache-Miss-Initiated Prefetching Techniques

In this section, we overview the tradeoffs involved in each of the cache-miss-initiated prefetching techniques we consider.

2.1 Large Cache Blocks

The choice of block size depends on the locality and sharing properties of applications, as well as the remote access latency and bandwidth. The spatial and processor (sharing) locality of applications determines how miss rates vary as a function of the block size. Applications with good spatial locality usually benefit from using larger cache blocks, since most of the data in a cache block is likely to be referenced before it is evicted or invalidated. In the absence of write sharing of data, an increase in the block size reduces the miss rate until the *cache pollution* point [Eggers and Katz, 1989].

The relationship between the cache block size and the miss rate has been studied extensively in the context of uniprocessors (e.g., [Przybylski, 1990]), but the miss rates of parallel programs do not always follow the same trends as sequential programs [Eggers and Katz, 1989]. Applications with coarse-grain sharing typically favor large cache blocks since, for these applications, the true sharing miss rate goes down with an increase in block size. Applications with fine-grain sharing usually favor small cache blocks, so as to avoid false sharing, and to avoid bringing data into the cache that will be invalidated before referenced.

In the best case (perfect spatial locality and coarse-grain sharing), doubling the size of cache blocks would cut the miss rate in half. Unfortunately, this best case scenario is extremely rare; increasing the block size typically causes more misses of one type while reducing the number of misses of another type.

The choice of block size does not depend solely on miss rates of applications; we must also consider architectural parameters. In particular, remote access latency and bandwidth are important factors, as they determine the cost of fetching a cache block.[1] High remote access latency favors large cache blocks, since more data can be accessed with the same latency penalty. High remote access bandwidth also favors large cache blocks, since more data can be transferred for little extra cost. Large cache blocks can introduce network contention problems however, since small packets generate less contention than large ones (assuming the same amount of data is transferred in both cases) [Agarwal, 1991]. Also, memory performance is affected by the block size; large blocks increase the memory busy time, thereby delaying contending processors.

Increased network and memory bandwidth can reduce the cost of transferring large cache blocks, but do not change the role of the miss rate. An increase in block size only improves performance when the larger blocks result in a lower miss rate. Even then, the decrease in the miss rate must be enough to offset the higher miss penalty of larger blocks.

Several researchers have studied the impact of cache block size on the miss rate and overall message traffic on small-scale, bus-based multiprocessors (e.g., [Eggers and Katz, 1989]). The results

[1]The latency of the memory is the time it takes to deliver the first word of data from the memory. The latency of the network is the time it takes to transfer a single word of data from source to destination. The bandwidth of the network (or memory) is the number of bytes transferred per second after the initial latency period.

of these studies do not apply directly to scalable, network-based machines however, which incorporate very different architectural tradeoffs. Other studies (e.g., [Dubnicki, 1993; Lee *et al.*, 1987]) have explored the relationship between block size and network bandwidth, but these studies either ignore one or more important factors (such as finite-sized caches, or network contention), or assume a different architecture. We addressed these concerns in [Bianchini and LeBlanc, 1994], where we studied the relationship between cache block size and application performance as a function of remote access bandwidth and latency.

2.2 Sequential Prefetching

Even when large cache blocks reduce the miss rate, the higher miss penalty may actually hurt overall performance. One way to reduce the miss penalty, while still retaining the potential for lower miss rates, is to use sequential prefetching with small cache blocks. Under sequential prefetching, a read miss causes some number of successive blocks to be prefetched independently.[2] Prefetches are only issued for blocks for which there are no pending operations, and which are not in the cache at the time of the miss. The processor can continue execution as soon as the block that caused the miss is loaded into the cache. In this way, the cost of prefetching other blocks can be overlapped with computation.

In terms of the read miss rate, sequential prefetching has the potential to perform well for programs that benefit from large cache blocks (i.e., programs with good spatial locality and limited sharing). In contrast to large cache blocks, sequential prefetching is less likely to suffer from false sharing, since the coherency units are small.

In the absence of false sharing, sequential prefetching generates more network transactions than large cache blocks, since several prefetch requests are required to load the data in a large cache block. Each request can be serviced more rapidly however, and requests from different processors may get interleaved. This latter feature is particularly important under tight synchronization constraints.

Although sequential prefetching has been studied extensively in the context of uniprocessors (e.g. [Smith, 1978]), the same is not true for multiprocessors. [Dahlgren *et al.*, 1993] compares the performance of sequential prefetching with an adaptive sequential prefetching technique for scalable multiprocessors. They also studied the performance of large cache blocks that fetch the same amount of data as the sequential prefetching strategy. Their results showed that adaptive prefetching performs at least as well as sequential prefetching, and that both strategies perform better than large cache blocks. This study assumed infinite network bandwidth however, and did not investigate how aggressively prefetches could be issued. In particular, their implementation of sequential prefetching only fetched a single extra block on a read miss.

2.3 Hybrid Prefetching

Both large cache blocks and sequential prefetching only work well for programs with very good spatial locality. Stride-directed prefetching [Fu and Patel, 1992] and lookahead data prefetching [Baer and Chen, 1991] are examples of prefetching techniques that attempt to deal with large strides in data accesses.

[2]Write misses and requests for exclusive access to shared data could also prefetch additional blocks, but we do not consider these cases. Write buffers and relaxed consistency are sufficient to hide write latencies in most cases.

Under stride-directed prefetching, a Stride Prediction Table (SPT) is used to store the last memory address referenced by an instruction. Strides are automatically computed by subtracting consecutive memory addresses referenced by an instruction. Once the stride of access for an instruction is computed, a prefetch of the next required memory block can be issued (provided that the stride is non-zero).

Hybrid prefetching is similar to stride-directed prefetching in that both use a hardware table for storing stride-related information. Hybrid prefetching uses an instruction/stride table (IST) indexed by instruction address, where each entry contains the number of blocks to prefetch on a read miss and a stride between the blocks. On a read miss, the cache controller fetches the block that caused the miss and prefetches additional blocks with a certain stride, as determined by the IST entry for the instruction. If the instruction has no corresponding entry in the IST, a single block (with stride 1) is prefetched on its behalf.

Under hybrid prefetching, the compiler computes the stride of access for an instruction and the number of blocks to prefetch on each cache miss, and generates code to fill in the IST. This code usually resides outside loops, and therefore the overhead of changing the table is negligible.

There are several differences between stride-directed prefetching and hybrid prefetching:

- Under stride-directed prefetching, the strides are generated on-the-fly using dynamic reference information, while the stride information is generated by the compiler under hybrid prefetching.
- Stride-directed prefetching only prefetches one block, while hybrid prefetching allows several blocks to be fetched on a read miss.
- Under hybrid prefetching different instructions can prefetch a different number of blocks, while under stride-directed prefetching all instructions prefetch the same number of blocks.
- The IST is managed by the compiler, so the table itself can be very small (i.e., 4 or 8 entries). The SPT requires a separate entry for each prefetching instruction, and therefore must be very large (possibly on the order of 1K entries).
- The SPT resides on the processor chip; the IST resides outside the processor chip, since we only prefetch on cache (read) misses. Thus, under hybrid prefetching, the instruction address must be available outside the processor chip on a read miss.

Like stride-directed prefetching, hybrid prefetching does not perform well with irregular strides. Another drawback of hybrid prefetching is that it depends on the compiler being able to determine strides of access for the relevant instructions in the program. When this analysis is not possible for a particular instruction, hybrid prefetching must either default to a less sophisticated prefetching strategy (e.g., sequential prefetching) or simply avoid prefetching for that instruction.

Hybrid prefetching is strictly more powerful than sequential prefetching, since the IST can be programmed to prefetch blocks with unit stride. In fact, it is easy to resort to sequential prefetching whenever the stride cannot be determined at compile time. In addition, the number of blocks to prefetch on a miss can be varied on a per instruction basis. Hybrid prefetching also compares favorably against large cache blocks, since hybrid prefetching not only performs well for large regular access strides, but also reduces the miss penalty by handling small cache blocks. The main disadvantage of hybrid prefetching is that it requires additional hardware (i.e., the IST).

The extent to which hybrid prefetching dominates the other cache-miss-initiated techniques depends on the stride of access in parallel programs. In the section 4, we describe the access patterns of our application suite, and evaluate the performance of each of the cache-miss-initiated prefetching strategies under varying assumptions about bandwidth.

3 Methodology and Workload

We are interested in exploring variations in bandwidth and prefetching strategies in scalable shared-memory multiprocessors, and therefore direct experimentation is not an available option. Thus, we use simulation for our studies.

3.1 Multiprocessor Simulation

We use an on-line, execution-driven simulator that exploits a mixture of interpretation and native execution to simulate unmodified MIPS R3000 object code. The simulator is divided into two parts, an event generator [Veenstra, 1993] and an event executor. The event generator simulates the processor and registers and calls the event executor on every memory reference. The event executor determines which processors block awaiting remote references and which processors continue to execute.

We simulate events at the level of processor cycles; all simulation parameters and results are expressed in terms of processor cycles. Our event executor deals with all the major components of a parallel computing system: caches, the interconnection network, local memories, and directories.

We simulate a scalable direct-connected multiprocessor with 32 nodes. Each node in the simulated machine contains a single processor, cache memory, local memory, directory memory, and a network interface. The connection between these node components is clocked at half the speed of the processor. Each processor has a 16-entry write buffer and a 128 KB direct-mapped, lock-up free, write-back cache. The cache block size is a parameter in our study. Caches are kept coherent using an implementation of the DASH protocol with release consistency [Lenoski *et al.*, 1990].

The simulator implements a full-map directory for controlling the state of each block of memory. Each node contains the directory for the memory associated with that node.

Throughout this paper we refer to the ensemble of addressable local memory and directory memory at each node as a "memory module." Shared memory is interleaved among the nodes at a memory (cache) block granularity, i.e. consecutive blocks are assigned to successive nodes in round-robin fashion.[3] Memory modules queue requests (coming either from the cache or network interface) when the module is busy. Memory queues are assumed to be infinite. The latency of a memory module is 24 processor cycles. The memory transfer rates we use are described in table 1 (assuming 100 MHz clocks).

The interconnection network is a bi-directional wormhole-routed mesh, with dimension-ordered routing. The network clock speed is the same as the processor clock speed. Switch nodes introduce

[3]This memory organization is used in other multiprocessors, including the BBN TC2000 [BBN, 1989].

Level	Latency	Cycles/Word	Memory Bandwidth
Infinite	24 cycles	0 cycles	Infinite
High	24 cycles	0.5 cycles	800 MB/sec
Medium	24 cycles	2 cycle	200 MB/sec
Low	24 cycles	4 cycles	100 MB/sec

Table 1: Memory bandwidth levels used in simulated machine.

Level	Path Width	Latency/Switch	Latency/Link	Bi-dir Link Bandwidth
Infinite	Infinite	5 cycles	1 cycle	Infinite
High	128 bits	5 cycles	1 cycle	3.2 GB/sec
Medium	32 bits	5 cycles	1 cycle	800 MB/sec
Low	16 bits	5 cycles	1 cycle	400 MB/sec

Table 2: Network bandwidth levels used in simulated machine.

a 5-cycle delay to the header of each message. The bandwidth of the network is a parameter in our study. In finite-bandwidth networks (derived from the Alewife cycle-by-cycle network simulator), contention for network links and buffers is fully captured. Each network interface has a queue for out-going messages, which is fed either by the cache or the memory module at the node. For comparison purposes we also implement an idealized, infinite bandwidth network, in which the path width is always larger than the size of messages. The idealized network only models contention at the source and destination of messages. The levels of network bandwidth we use are described in table 2 (again, assuming 100 MHz clocks).

3.2 Performance Metrics

For the most part our focus is on three different metrics: the read miss rate, the memory access stall time per processor, and the running time of the application. We ignore write misses in most cases because we assume deep write buffers and release consistency, which serve to hide the cost of writes. The read miss rate is computed solely with respect to shared references. That is, the read miss rate is defined as the total number of read misses on shared data divided by the total number of reads to shared data. We classify misses using an extension of the algorithm in [Dubois *et al.*, 1993].

The memory access stall time (MAST) is defined as the total stall time experienced by all processors due to memory references (read misses and stalls caused by a full write buffer) divided by the number of processors. In most cases, read misses account for almost all of the stall time; unless stated otherwise, write overheads are negligible.

Using running time as a metric accounts for all activities that occur during the simulated execution of a program. Accesses to code and private data are modeled as cache hits.

Application	Shared Refs	Shared Reads (% of shared refs)	Shared Writes (% of shared refs)
`Barnes-Hut`	54.7 M	98 %	2 %
`Gauss`	64.5 M	67 %	33 %
`MMp3d`	12.7 M	64 %	36 %
`Blocked LU`	47.3 M	90 %	10 %

Table 3: Memory reference characteristics on 32 processors.

3.3 Workload

Our application workload consists of four parallel programs: `Barnes-Hut`, `MMp3d`, `Blocked LU`, and `Gauss`. `Barnes-Hut` is an N-body application that simulates the evolution of 4K bodies under the influence of gravitational forces for 4 time steps. `MMp3d` is an improved version of `Mp3d`, a wind-tunnel airflow simulation of 30000 particles for 20 time steps. Both `Barnes-Hut` and `Mp3d` are from the SPLASH suite [Singh *et al.*, 1992]. In our modified implementation of `Mp3d`, particles are assigned to processors in such a way as to reduce sharing significantly. `Blocked LU` performs blocked right-looking LU decomposition [Dackland *et al.*, 1992] on a 384 $\times$ 384 matrix. `Gauss` is an unblocked implementation of Gaussian elimination on a 400 $\times$ 400 matrix. Table 3 summarizes the distribution of shared references in our applications on a 32-processor machine.

As is the case with similar studies, simulation constraints prevent experimentation with "real life" input data sets. Simply reducing the input size to manageable levels without changing the cache size could produce unrealistic results however. Therefore the input data sizes used for our applications were chosen in tandem with our choice of cache size. We first determined input sizes that could be simulated in a reasonable amount of time, and then experimented with various cache sizes for those data sets. The cache size we ultimately selected, 128 KB, was chosen so as to avoid too heavy an emphasis on replacement misses; this cache size is the smallest that holds the working set of processors for our applications.

4 Performance Evaluation of Prefetching Strategies

In this section, we evaluate the performance of three cache-miss-initiated prefetching strategies. We first explore the effect of large cache blocks on the read miss rate and the memory access stall time (MAST) of our application suite. We then investigate the effect of sequential prefetching and hybrid prefetching on the miss rate and MAST as we vary the number of blocks prefetched on a read miss. Finally, we examine the overall effect on running time of each of the cache-miss-initiated prefetching techniques, and compare them to software prefetching, which does not require misses to initiate prefetching.

4.1 Large Cache Blocks

Assuming that write buffers and release consistency can hide the cost of writes, then the block size that results in the minimum read miss rate represents an upper bound on the effective size

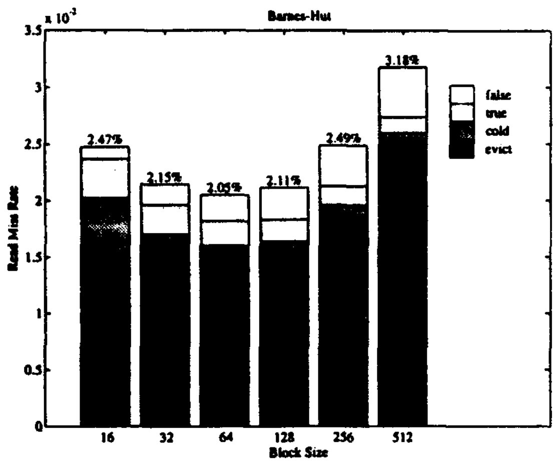

Figure 1: Read miss rate of Barnes-Hut.

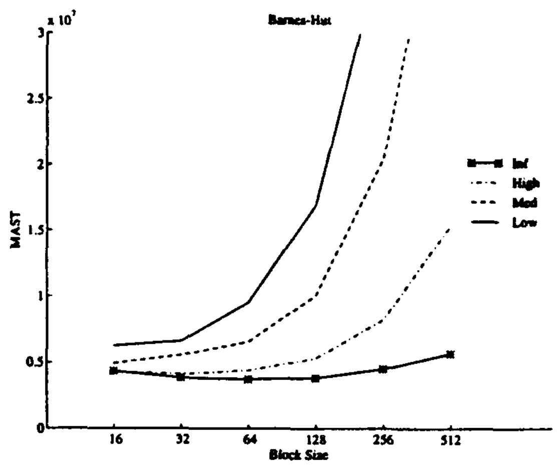

Figure 2: MAST of Barnes-Hut.

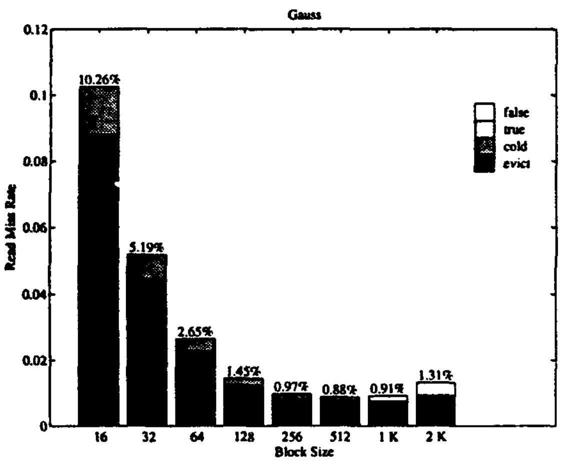

Figure 3: Read miss rate of Gauss.

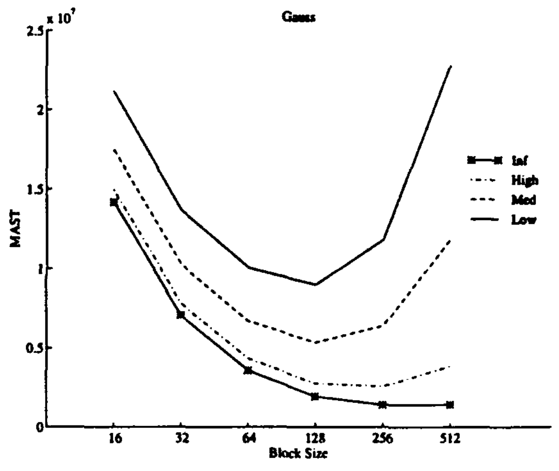

Figure 4: MAST of Gauss.

of cache blocks. Larger blocks simply increase the MAST (and consequently the running time) of the application, regardless of the available bandwidth or the remote access latency. Given infinite bandwidth, the block size that minimizes the read miss rate is optimal in terms of the overall remote access cost; smaller blocks incur larger penalties for transferring the same amount of data.

Our ability to hide the cost of writes depends on the block size however. Increasing the block size may increase the cost of write operations (and synchronization latency) due to the resulting higher degree of sharing. Thus, even under infinite bandwidth, the block size that minimizes the read miss rate may not produce the minimum stall time.

Figures 1-8 present the read miss rates and MASTs for each of our applications as a function of cache block size. In the miss rate figures, the percentage at the top of each column represents the percent of all reads to shared data that result in a miss; within a column misses are classified as either eviction, cold start, true sharing, or false sharing misses.

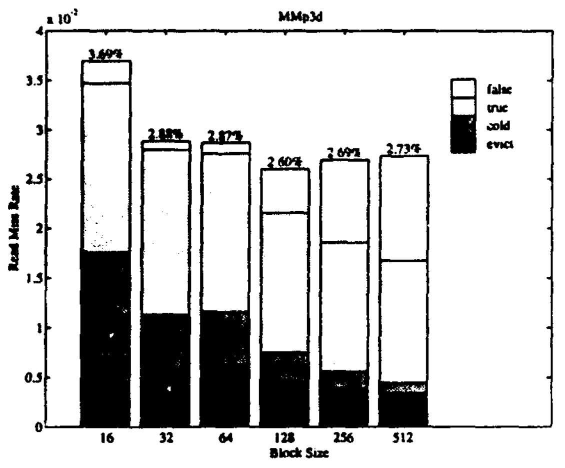

Figure 5: Read miss rate of `MMp3d`.

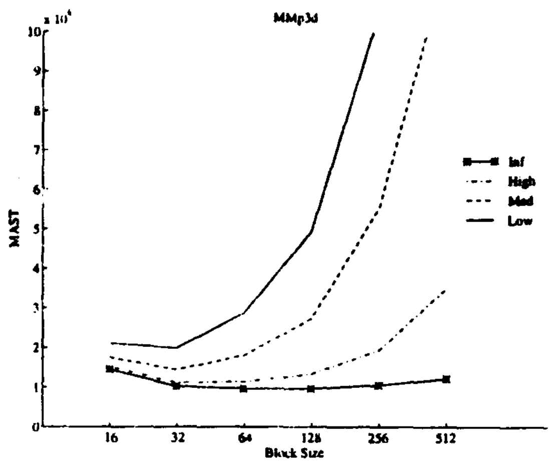

Figure 6: MAST of `MMp3d`.

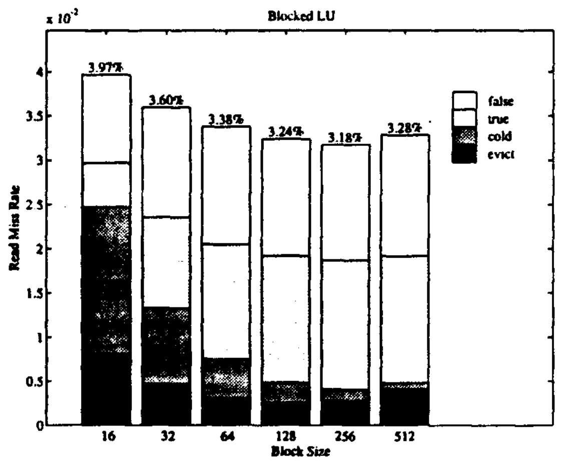

Figure 7: Read miss rate of `Blocked LU`.

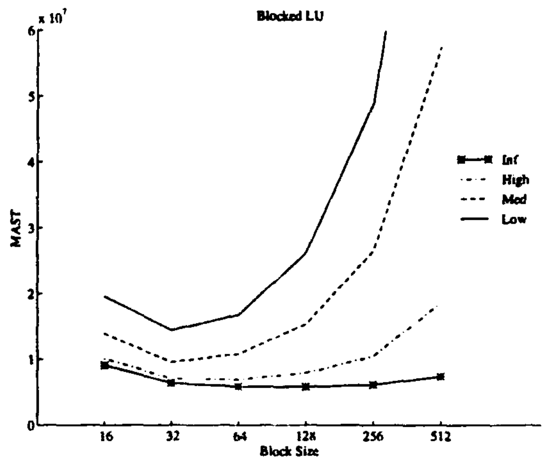

Figure 8: MAST of `Blocked LU`.

Figure 1 shows the read miss behavior of Barnes-Hut. Even though the working set of a processor fits in its cache, evictions are still a problem due to limited spatial locality and to the mapping of addresses in direct-mapped caches. The minimum read miss rate occurs with 64-byte blocks; larger blocks increase the number of eviction misses (due to cache pollution) and false sharing misses. The other categories of misses decrease with an increase in block size.

Although increasing the block size up to 64 bytes decreases the read miss rate, figure 2 shows that the MAST is minimized with 16-byte blocks for most practical levels of bandwidth. The improvements in miss rate for larger blocks are not enough to justify the increased miss penalty. However, under infinite bandwidth 32, 64, and 128-byte blocks perform best; these block sizes offer the lowest read miss rate (around 2.1%) and comparable read miss penalties (around 105 cycles).

Figure 3 shows the miss behavior of Gauss. As with Barnes-Hut, the miss rate of Gauss is dominated by cache evictions. Evictions in Gauss are caused by poor temporal locality among accesses to the main matrix; each processor repeatedly references a large portion of the matrix for each row it is updating. Repeatedly doubling the block size (up through 128 bytes) continually cuts the miss rate roughly in half. These improvements in the read miss rate are due to the excellent spatial and processor locality of the program. Beyond 128-byte blocks, the read miss rate improves much more slowly, with the minimum miss rate occurring when the block size is 512 bytes. Evictions and false sharing increase the read miss rate when increasing the block size beyond 512 bytes.

Figure 4 demonstrates that increasing the block size to 128 bytes significantly reduces the MAST for Gauss, regardless of the available bandwidth. However, for the finite levels of bandwidth, increases in the block size beyond 128 bytes do not reduce the MAST, even though the read miss rate is minimized at 512-byte blocks. The read miss penalty for 512-byte blocks is simply too high: 1900, 980, and 320 processor cycles for low, medium, and high bandwidth levels, respectively. 512-byte blocks do perform best with infinite bandwidth, where a small reduction in miss rate is enough to offset a minor increase in miss penalty (to 120 cycles).

As seen in figure 5, increasing the block size up to 128 bytes results in a decrease in the read miss rate of MMp3d. Although this trend in the miss rate is similar to the trends for Barnes-Hut and Gauss, the composition of the miss rate for MMp3d is markedly different. For MMp3d, the read miss rate is dominated by sharing-related misses instead of evictions. False sharing is the limiting factor that precludes the use of 256-byte blocks.

Figure 6 presents the MAST of MMp3d. For the low and medium levels of bandwidth, performance suffers when using 128-byte blocks, even though this block size produces the minimum read miss rate. The improvement in read miss rate offered by 128-byte blocks over 64-byte blocks does not offset the increase in the read miss penalty, particularly at lower levels of bandwidth, where miss penalties increase from 245 to 390 cycles under medium bandwidth and from 380 to 690 cycles under low bandwidth. Even with high bandwidth, a fairly small block size (32 bytes) performs as well as, or better than, larger block sizes.

In this case, the excessive memory access stall time produced by large blocks is due primarily to the failure of the write buffer to hide write costs. Large blocks result in more apparent sharing behavior, and hence a greater chance that a write operation will stall the processor. Thus, although writes account for only 10% of the stall time with 128-byte blocks and medium bandwidth, they account for 25% of the stall time with 512-byte blocks and medium bandwidth.

Figure 7 presents the miss rate behavior of Blocked LU. As with MMp3d, sharing-related misses dominate the read miss rate when the block size is larger than 16 bytes. For the first time, we see

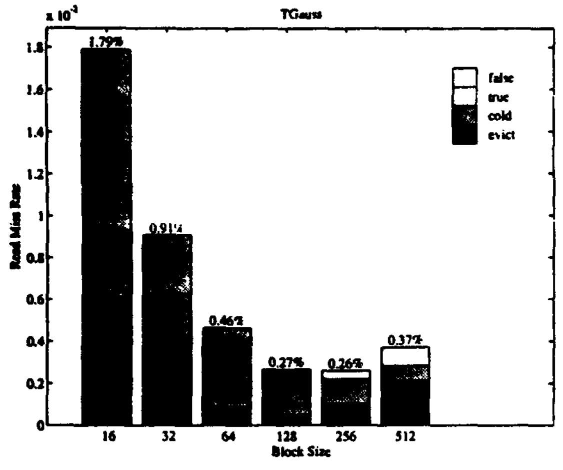

Figure 9: Read miss rate of TGauss.

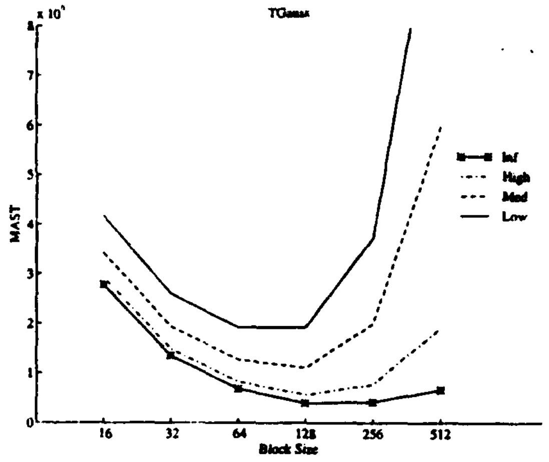

Figure 10: MAST of TGauss.

significant amounts of false sharing introduced with relatively small cache blocks. Despite the false sharing, the minimum miss rate is achieved with large cache blocks (256 bytes).

As seen in figure 8, Blocked LU and MMp3d have similar MAST behavior. That is, the block size that minimizes the read miss rate (256 bytes) performs much worse than smaller block sizes at the lower levels of bandwidth. Even under infinite bandwidth, most of the performance gains achievable by increasing the block size are captured by a fairly small cache block size (32 bytes).

To see whether more carefully tuned application programs can exploit larger cache blocks, we modified Gauss to improve its temporal locality, and thereby reduce the number of eviction misses. We modified the program so that each processor reads a pivot row once, updates all of its local rows based on that pivot row, and then reads the next pivot row. The resulting program is called TGauss.

By comparing the miss rates of Gauss (figure 3) and TGauss (figure 9) we can see that this modification is very successful at reducing the number of replacement misses. The overall read miss rate of TGauss is nearly a factor of 6 smaller than the read miss rate of Gauss for most block sizes. It is therefore surprising to see that the minimum read miss rate for TGauss occurs with 128 and 256-byte blocks, whereas the minimum read miss rate for Gauss occurs with 512-byte blocks. The composition of misses is different for the two programs, although evictions are the main driving force in the overall read miss rate in both cases.

Although the upper limit on effective block size for TGauss is smaller than the upper limit for Gauss (128 vs. 512 bytes), both programs achieve their lowest MAST with 128-byte cache blocks in most cases. Thus, in this case, a program modification that significantly improves locality does not increase the size of cache blocks that can be utilized effectively.

From these examples it is clear that several factors contribute to the read miss rate of applications, any one of which can limit effective increases in the block size. Bandwidth limitations further constrain the effective size of cache blocks. For our applications, block sizes between 16 and 128 bytes provide the best MAST under medium and low bandwidth, while block sizes between 32 and 256 bytes perform best with high bandwidth. Fairly small cache blocks (32 or 64 bytes) can achieve most of the performance benefits of larger blocks, even under infinite bandwidth, because larger

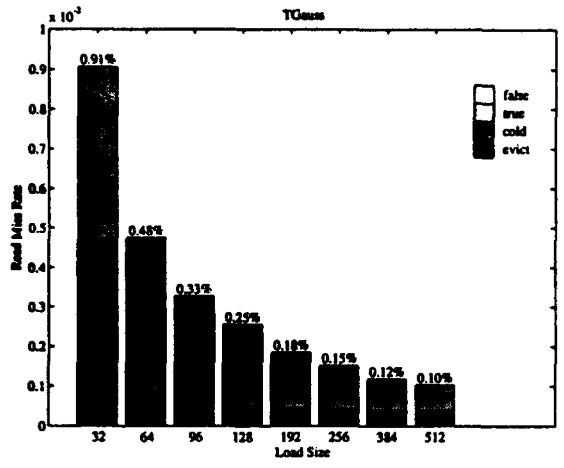

Figure 11: Read miss rate of TGauss under sequential prefetching.

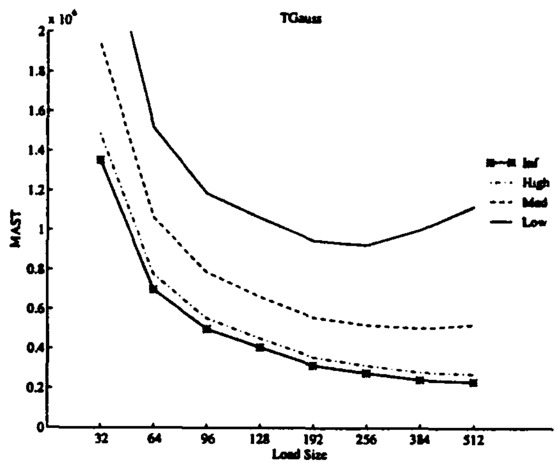

Figure 12: MAST of TGauss under sequential prefetching.

blocks reduce the miss rate by only a marginal amount. Furthermore, improvements in locality of reference may not translate to effective increases in the block size.

See [Bianchini and LeBlanc, 1994] for a complete and detailed analysis of the effect of block size on the miss rate and application performance.

4.2 Sequential Prefetching

In the previous section we saw that increasing the block size can drive up the miss rate or dramatically increase the miss penalty, which precludes the use of large cache blocks as an effective prefetching technique. In this section, we investigate whether sequential prefetching can do better, by alleviating the false sharing and high miss penalties associated with large blocks. Our investigation of sequential prefetching is based on three programs: TGauss, MMp3d, and Blocked LU.

Figures 11-16 present the read miss rate (under infinite bandwidth) and the MAST of our three applications as a function of the load size (that is, the total number of bytes fetched and prefetched on a read miss) under sequential prefetching. We use 32-byte cache blocks, and vary the number of blocks prefetched on a miss.

Figure 11 shows that sequential prefetching produces lower miss rates than large cache blocks for comparable load sizes. Under sequential prefetching the minimum read miss rate is only 0.10% (with a load size of 512 bytes), which is a factor of 3 smaller than the minimum read miss rate without sequential prefetching. Sequential prefetching performs better with the larger load sizes because it eliminates false sharing misses and reduces the eviction miss rate substantially.

As seen in figure 12, sequential prefetching also produces lower stall times than large cache blocks, primarily due to a decrease in the read miss penalty. For example, under sequential prefetching, the read miss penalties for a load of size 128 bytes are 220, 170, and 125 cycles for low, medium, and high bandwidth, respectively. The miss penalties for 128-byte cache blocks are 530, 310, and 160 cycles respectively.

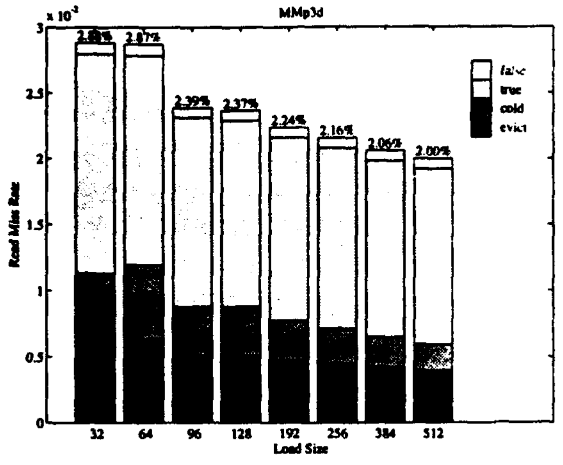

Figure 13: Read miss rate of MMp3d under sequential prefetching.

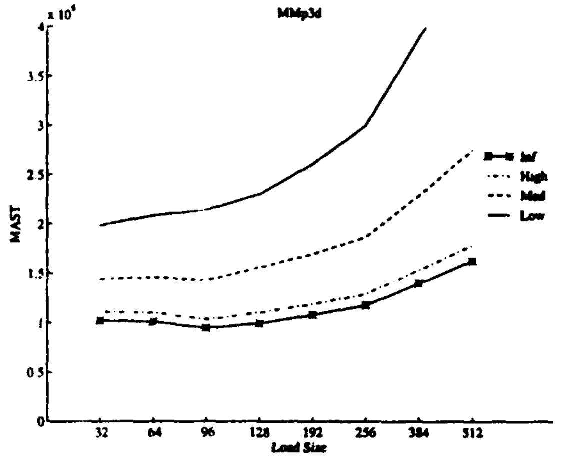

Figure 14: MAST of MMp3d under sequential prefetching.

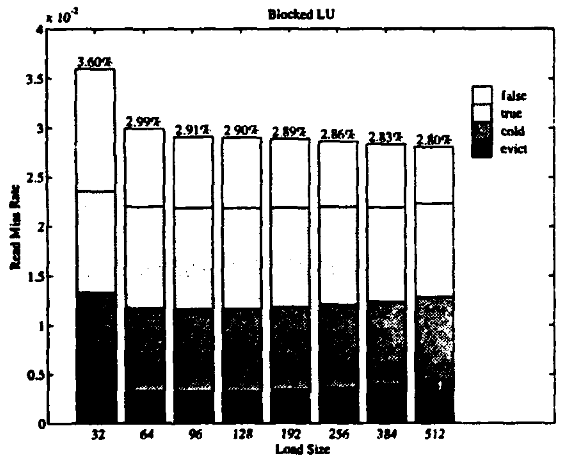

Figure 15: Read miss rate of Blocked LU under sequential prefetching.

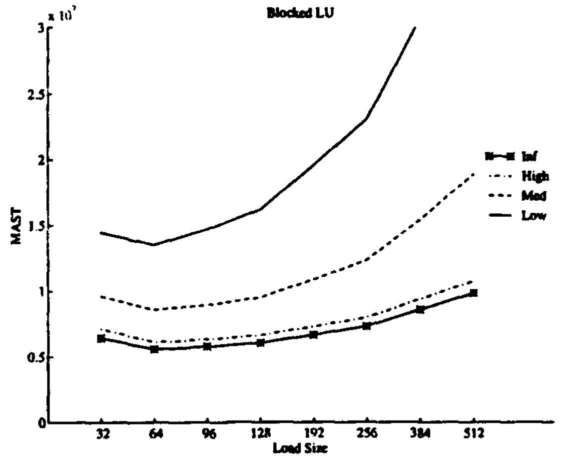

Figure 16: MAST of Blocked LU under sequential prefetching.

Figure 12 also shows that an increase in bandwidth allows more aggressive prefetching. For example, under low bandwidth, 256 bytes is the largest load size that reduces the MAST, but 384 bytes can be effectively utilized given medium or high bandwidth.

As seen in figure 13, the read miss rate of `MMp3d` is also reduced by sequential prefetching. The minimum read miss rate is reduced from 2.6% to 2.0%. This improvement in the miss rate is due almost entirely to fewer false sharing misses.

Although larger load sizes reduce the read miss rate under sequential prefetching, figure 14 shows that these improvements in the miss rate may not translate to reductions in the stall time. At the lowest level of bandwidth, a load size of 32 bytes produces the lowest M [illegible] Higher bandwidth allows more aggressive prefetching (up to 96 bytes), but even infinite banc [illegible] cannot justify the use of 512-byte loads, even though this load size produces the lowest read [illegible] rate. The problem with large load sizes for `MMp3d` is that the cost of writes begins to dominate performance; writes account for as much as 45% of the MAST with a 512-byte load size.

`Blocked LU` exhibits the smallest improvement in read miss rates from sequential prefetching. The minimum miss rate produced by sequential prefetching is only 13% lower than the minimum miss rate achieved without sequential prefetching. Most of the improvement comes fr[illegible]m a reduction in false sharing misses.

Figure 16 shows that the MAST of `Blocked LU` is minimized with a load size of 64 bytes at all levels of bandwidth. Once again, writes account for a large portion of the MAST, especially at the larger load sizes. For example, with a 512-byte load size and low bandwidth, writes account for 30% of the stall time.

In summary, while aggressive sequential prefetching often improves the read miss rate of applications, it may not significantly reduce the stall time. In the particular case of programs with fine-grain sharing and lots of write operations (e.g., `MMp3d`), the minor improvements in the read miss penalty offered by aggressive prefetching may not compensate for a corresponding increase in the cost of writes.

4.3 Hybrid Prefetching

In this section, we investigate whether hybrid prefetching can improve on the performance of sequential prefetching. Hybrid prefetching has the potential to perform better, since it can prefetch with *any* fixed stride between blocks, while being selective about how aggressively to prefetch. In particular, the compiler may select aggressive prefetching for instructions dominated by cold misses, while using more conservative prefetching strategies for instructions that reference data that exhibit fine-grained sharing.

Our implementation of hybrid prefetching does not involve modifications to a real compiler. Instead, we collect the required stride information by profiling our programs during a simulation run. That is, we record a trace of the instructions with the highest miss rates and the addresses referenced by those instructions. For each instruction that generates a substantial number of misses, we process the trace to obtain the stride of access between every two consecutive references. We select the stride that occurs most frequently (and represents at least 25% of all references generated by the instruction) as the prefetching stride. Using this information, we manually instrument our programs with directives for modifying the instruction/stride table at run time. The number of

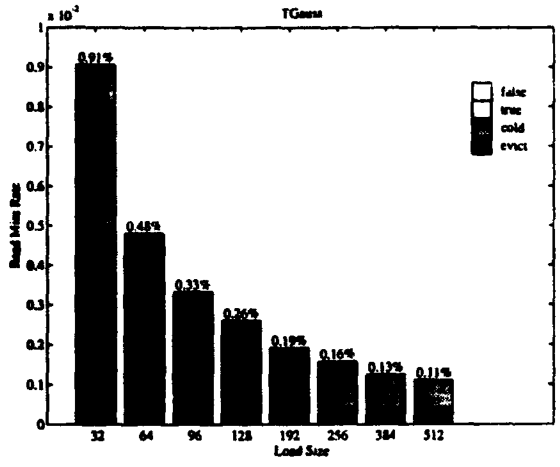

Figure 17: Read miss rate of TGauss under hybrid prefetching.

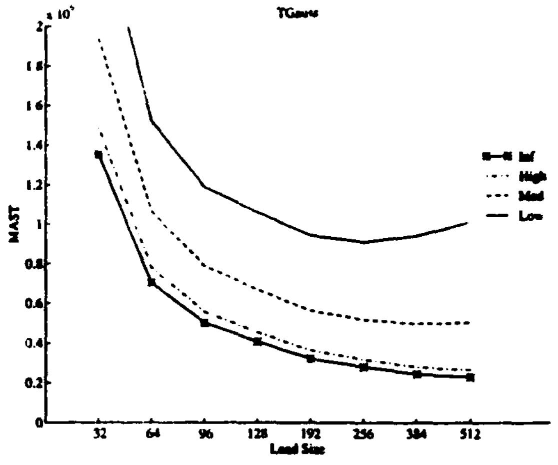

Figure 18: MAST of TGauss under hybrid prefetching.

blocks to prefetch is constant for all instructions (except in a few cases, where we limit prefetching to a single block due to fine-grain sharing), and we vary this number as one of the parameters for our study.

The performance of hybrid prefetching is dictated in large part by the access stride (in terms of cache blocks) and sharing patterns of applications. Both Gauss and TGauss exhibit unit stride, since most of the references in their inner loops are to consecutive cache blocks. Thus, both sequential and hybrid prefetching can be effective for Gauss and TGauss. In MMp3d, accesses to consecutive particles and adjacent (along the x axis) space cells result in references to alternating cache blocks, since these data structures are padded to fill two cache blocks. For Blocked LU the most common stride of access is 48 blocks, which is caused by accesses to columns in a matrix stored in row-major order. We would expect both MMp3d and Blocked LU to benefit from hybrid prefetching, since both of these programs have a significant fraction of references with a fixed, non-unit stride.[4] For Barnes-Hut, the vast majority of accesses in the program have no regular stride, so this program is likely to pose serious problems for any prefetching strategy that depends on regular access patterns.

The sharing behavior of MMp3d and Blocked LU is similar in that both programs exhibit fine-grain sharing; the miss rate of both applications is dominated by true and false sharing misses. In the case of MMp3d, most of the misses (40%) are the result of true sharing of the data structure representing the wind tunnel space (the cell array). For Blocked LU, most of the misses are caused by false sharing of data in the main matrix. As a result, our implementation of hybrid prefetching only prefetches one additional block when satisfying a read miss to either of these data structures.

Figures 17-22 show the read miss rate and MAST of TGauss, MMp3d, and Blocked LU under hybrid prefetching. As expected, hybrid prefetching and sequential prefetching perform exactly the same for TGauss. For MMp3d hybrid prefetching produces slightly lower miss rates than sequential

[4]As seen in the previous section, sequential prefetching with a load size of 64 bytes is not particularly effective for MMp3d, but prefetching with a load size of 96 bytes is effective. The explanation for this behavior is the dominant number of references to alternating cache blocks in MMp3d. In this case, prefetching one block (a load size of 64 bytes) does not help, but prefetching two blocks does help.

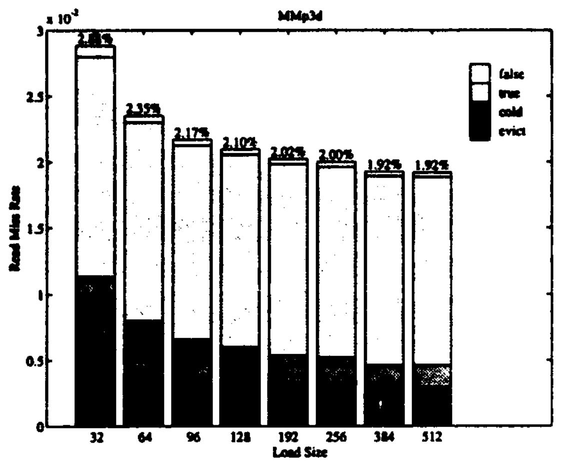

Figure 19: Read miss rate of MMp3d under hybrid prefetching.

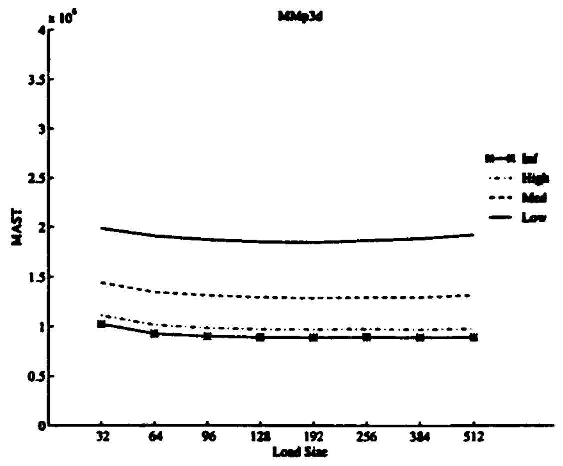

Figure 20: MAST of MMp3d under hybrid prefetching.

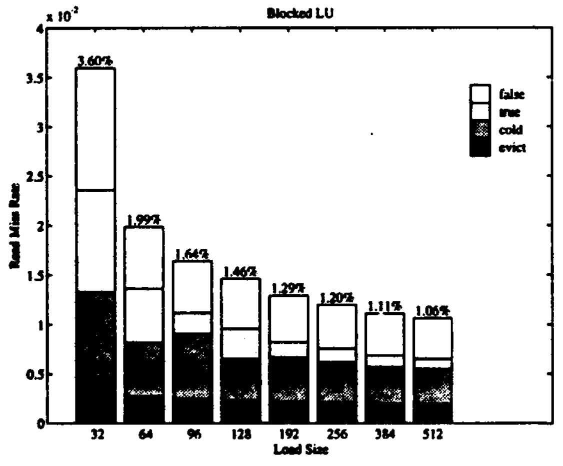

Figure 21: Read miss rate of Blocked LU under hybrid prefetching.

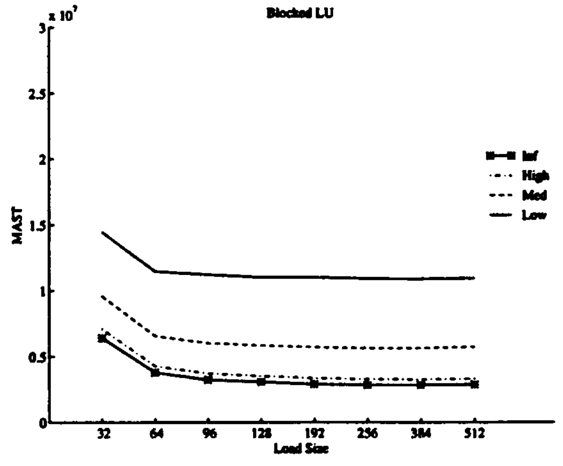

Figure 22: MAST of Blocked LU under hybrid prefetching.

prefetching, and a much lower MAST with large load sizes. Hybrid prefetching offers the most improvement for Blocked LU, cutting the minimum read miss rate from 2.8% to 1%, and substantially reducing the minimum MAST achievable.

By comparing figures 14 and 20 we observe that the MAST incurred by hybrid prefetching is relatively insensitive to load size, while the stall time incurred by sequential prefetching increases dramatically with load size. Unlike hybrid prefetching, the stall time produced by aggressive sequential prefetching is heavily influenced by write buffer stalls. The main reason write stalls are kept under control in hybrid prefetching is that we use a conservative prefetching strategy on the cell array in MMp3d, while simultaneously using an aggressive strategy on other data structures. Although our conservative strategy results in a slightly higher miss rate than would otherwise be possible with hybrid prefetching, it avoids the excessive stall time due to writes encountered under sequential prefetching.

By comparing figures 16 and 22 we can see that hybrid prefetching not only avoids excessive stall time due to writes, it also reduces the minimum MAST for each level of bandwidth. For Blocked LU, the minimum MAST produced by sequential prefetching under low bandwidth is 13.5M cycles, while the minimum MAST under hybrid prefetching is 10.9M cycles. Hybrid prefetching is even better under high bandwidth, decreasing the minimum MAST from 6.1M cycles to 3.2M cycles. In this case, the performance gap between hybrid and sequential prefetching increases with bandwidth, because higher bandwidth allows for more aggressive prefetching, which is beneficial (i.e., lowers the miss rate) in the case of hybrid prefetching, but not in the case of sequential prefetching (which can only benefit unit-stride accesses). Note that these improvements depend on a conservative implementation of hybrid prefetching for a select group of instructions; we only prefetch a single block on each read miss caused by any of five instructions, which together are responsible for 33% of the read misses in the program.

Blocked LU is an example of a program that exploits both of the properties that distinguish hybrid prefetching from sequential prefetching: a non-unit stride of access and a mixture of aggressive and conservative prefetching. The benefits of hybrid prefetching are limited however, because we avoid aggressive prefetching on five instructions that account for one third of the read misses. Perhaps by restructuring the program to reduce sharing, we could use aggressive prefetching on all the instructions, and reduce stall time even more.

To test this hypothesis, we modified Blocked LU to produce a new program called static blocked LU (SBlocked LU). In the modified program, each process works on a single set of data elements throughout its lifetime, rather than migrating among data elements in the interests of load balancing. Although SBlocked LU exhibits less sharing than Blocked LU, it does not keep all processors busy throughout the execution; on average, a processor drops out of the computation every three phases of the program.

Figure 23 shows the read miss rates of SBlocked LU. The minimum read miss rate produced by hybrid prefetching is 0.39%, which is a factor of 7 improvement over 32-byte blocks, and a factor of 5 improvement over the minimum read miss rate produced by sequential prefetching. Most of the improvements in the miss rate come from reductions in true and false sharing. As expected, SBlocked LU has lower miss rates than Blocked LU; the lowest miss rate of SBlocked LU is a factor of 2.7 smaller than the lowest miss rate of Blocked LU. More importantly, the improvement in the miss rate offered by prefetching is dramatically higher for SBlocked LU, a factor of 7 vs. a factor of 3.4 for Blocked LU.

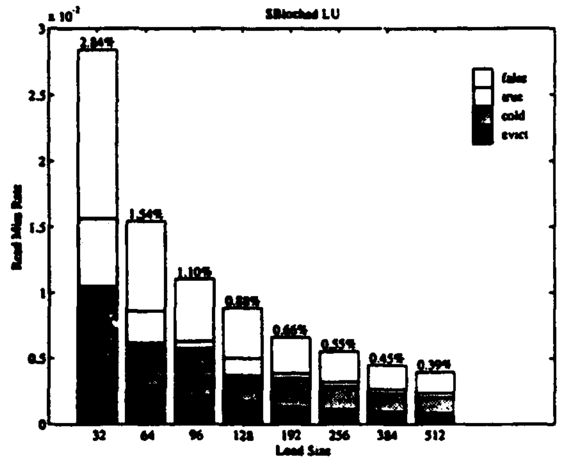

Figure 23: Read miss rate of SBlocked LU under hybrid prefetching.

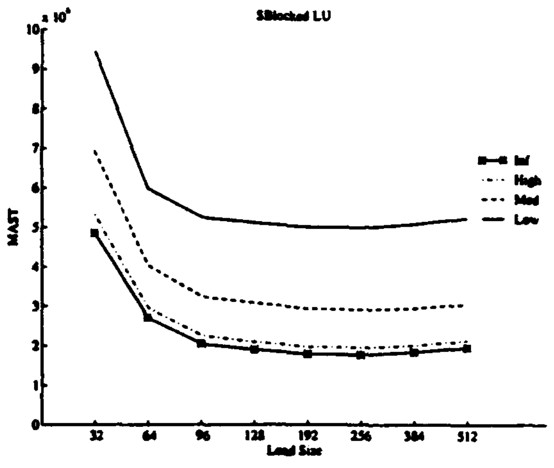

Figure 24: MAST of SBlocked LU under hybrid prefetching.

Figure 24 shows the MAST of SBlocked LU as a function of the load size and available bandwidth. As seen in the figure, there is a significant drop in the cost of memory accesses as we increase the load size up to 96 bytes, beyond which the MAST performance improves very slowly. Nonetheless, hybrid prefetching performs much better than sequential prefetching, lowering the minimum MAST of sequential prefetching by 41% under low bandwidth, and 53% under high bandwidth. In contrast, hybrid prefetching lowered the minimum MAST of sequential prefetching for Blocked LU with low bandwidth by only 19%, and by 48% with high bandwidth. This example illustrates the enormous benefits of aggressive prefetching in the absence of fine-grain sharing.

In summary, hybrid prefetching is comparable to sequential prefetching for programs with unit-stride access (such as TGauss), but offers additional opportunities for prefetching for programs with large, regular stride accesses (such as SBlocked LU). By using a mixture of aggressive and conservative prefetching within a program, hybrid prefetching can offer the benefits of prefetching for instructions with a regular access pattern, while avoiding prefetching on instructions that result in excessive sharing (as in MMp3d and Blocked LU). Since hybrid prefetching need not use the same load size on every instruction, it is better able to translate an increase in bandwidth to an increase in load size. As a result, the benefits of hybrid prefetching relative to sequential prefetching tend to increase with bandwidth.

4.4 Comparison of Prefetching Techniques

In this section we evaluate the success of cache-miss-initiated prefetching by examining the overall effect on running time of each technique. We also compare cache-miss-initiated prefetching with software prefetching [Callahan *et al.*, 1991; Mowry *et al.*, 1992], which does not require misses to initiate prefetching.

As with hybrid prefetching, we implemented software prefetching by hand. We use the miss rate information gathered for hybrid prefetching to determine the instructions that can benefit from prefetching. After identifying the most important instructions, we manually inserted prefetches so that data blocks are received just before they are required. In order to hide the latency of prefetching without generating substantial instruction execution overhead, we perform loop unrolling and splitting wherever necessary. Each prefetch instruction prefetches a single cache block in read mode; that is, we do not implement block and exclusive prefetches.

Figures 25-32 present a comparison of the running time produced by each of the techniques under low and high bandwidth assumptions for each of our applications. In these figures each column represents a different prefetching technique: 32-byte blocks with no prefetching (32B), the best block size for a given program (i.e., the block size that produces the smallest running time) with no prefetching (BBS), sequential prefetching with the load size that produces the smallest running time (SP), hybrid prefetching with the load size that produces the smallest running time (HP), and software prefetching (SWP). The number on the top of each column is the running time of that technique as a percentage of the running time for the base case of 32-byte blocks with no prefetching. Within a column running time is broken into busy time, read stall time, write stall time, and synchronization overhead.

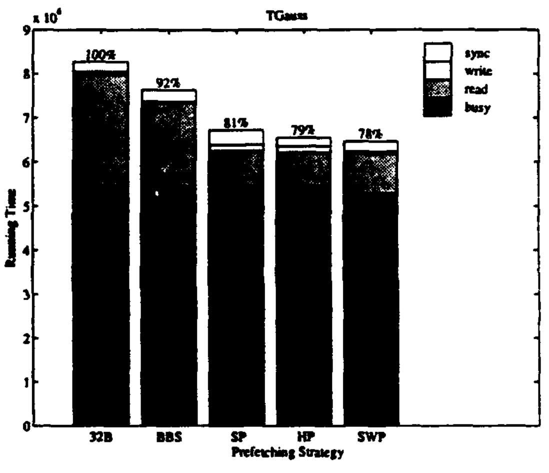

Figure 25: Running time of TGauss with low bandwidth.

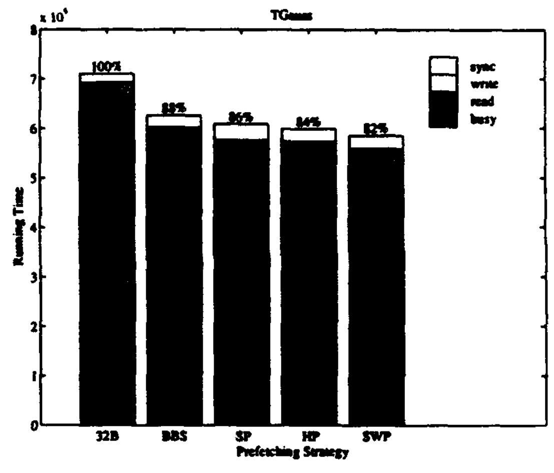

Figure 26: Running time of TGauss with high bandwidth.

Figure 25 shows the running time of TGauss with low bandwidth. As seen in the figure, sequential, hybrid, and software prefetching perform roughly the same for this program, improving its running time by about 20%. Using the best block size for this program (64 bytes) and no prefetching improves performance by only 8%.

It is interesting to note that software prefetching has a slightly lower busy time than the other techniques, despite the need for prefetching instructions. The reason for this counter-intuitive behavior is that in our implementation of software prefetching we unrolled the main computational loop in TGauss several iterations more than a standard compiler would have done. Without this aggressive unrolling, software prefetching introduces high instruction overhead.

In terms of the read stall overhead, software prefetching performed slightly worse than sequential and hybrid prefetching since, at this level of bandwidth, more than 30% of the prefetches issued under software prefetching are completed too late to avoid a miss.

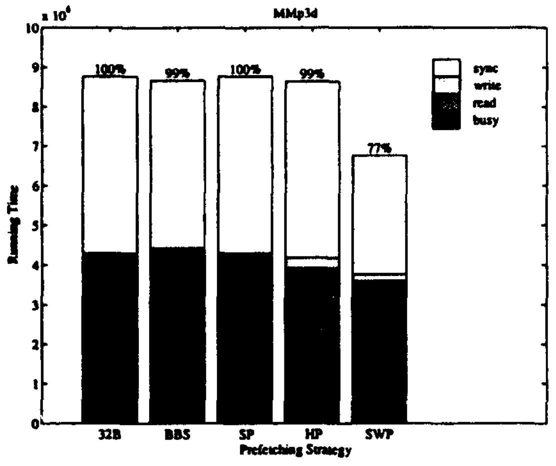

Figure 27: Running time of MMp3d with low bandwidth.

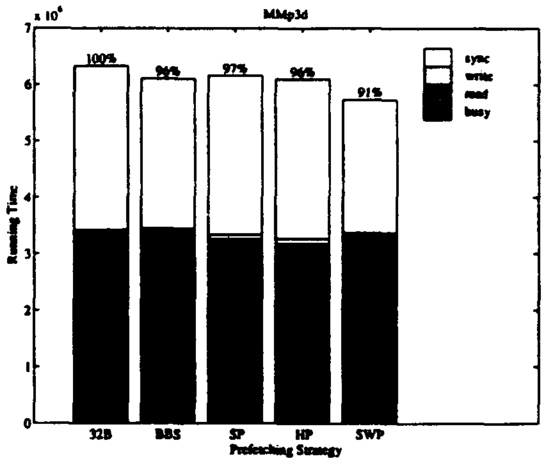

Figure 28: Running time of MMp3d with high bandwidth.

Figure 26 shows running times for TGauss under high bandwidth. At this level of bandwidth, the best block size (128 bytes) performed almost as well as the other techniques. Software prefetching performs better in terms of read stall time under high bandwidth, where only 9% of the prefetches are received late.

Figures 27 and 28 present running times for MMp3d under low and high bandwidth, respectively. In both figures we see that the cache-miss-initiated prefetching techniques are not successful at reducing the execution time significantly in comparison to the base architecture. Software prefetching, on the other hand, substantially improves running time (especially under low bandwidth), even though it increases the busy time.

As seen in figures 29 and 30, both hybrid prefetching and software prefetching are able to improve the running time of Blocked LU in comparison to the other techniques. Software prefetching performs better regardless of bandwidth because our implementation of hybrid prefetching is conservative on certain instructions. This conservative approach produces a higher read miss rate for hybrid prefetching (2.1% vs 1.6% at low bandwidth, and 1.5% vs 0.7% at high bandwidth).

Finally, figures 31 and 32 show the running time of SBlocked LU for each of the techniques. As with Blocked LU, both hybrid and software prefetching perform much better than the other techniques. Since SBlocked LU admits more aggressive cache-miss-initiated prefetching, hybrid and software prefetching offer comparable performance. Large block sizes and sequential prefetching produce very limited performance improvements, whereas hybrid and software prefetching improve the execution time by 20% to 24%, depending on bandwidth.

These results illustrate the circumstances under which cache-miss-initiated prefetching is most effective. If programs exhibit a regular access pattern, then each miss can prefetch a lot of data, and avoid future misses. Among the cache-miss-initiated techniques, only hybrid prefetching can adapt to large stride access patterns, and can tailor the amount of data prefetched on a miss according to the sharing behavior of each instruction. In the best case, hybrid prefetching offers performance comparable to software prefetching, which need not wait for a miss before issuing prefetches.

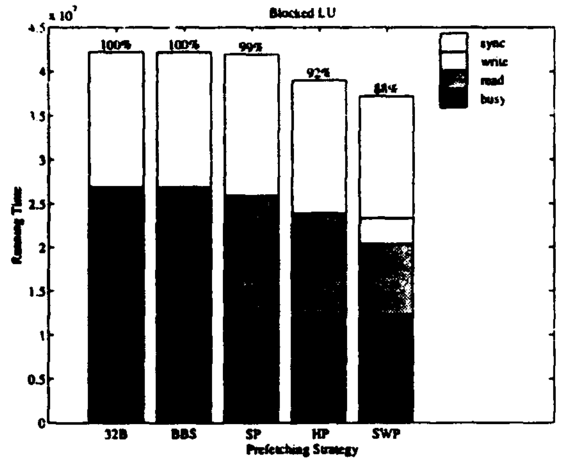

Figure 29: Running time of Blocked LU with low bandwidth.

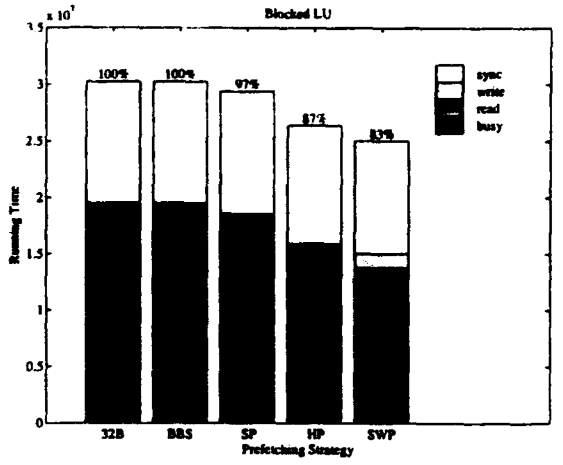

Figure 30: Running time of Blocked LU with high bandwidth.

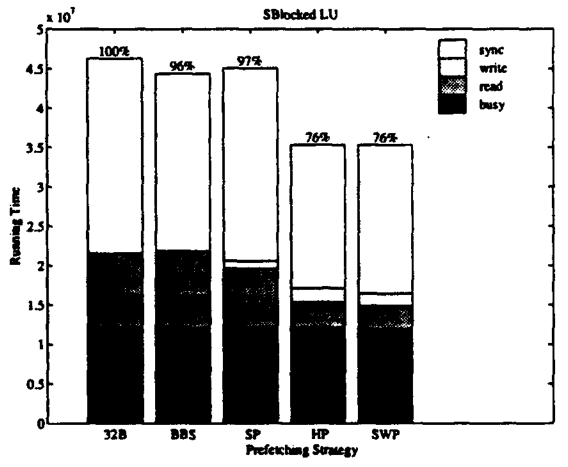

Figure 31: Running time comparison of SBlocked LU under low bandwidth.

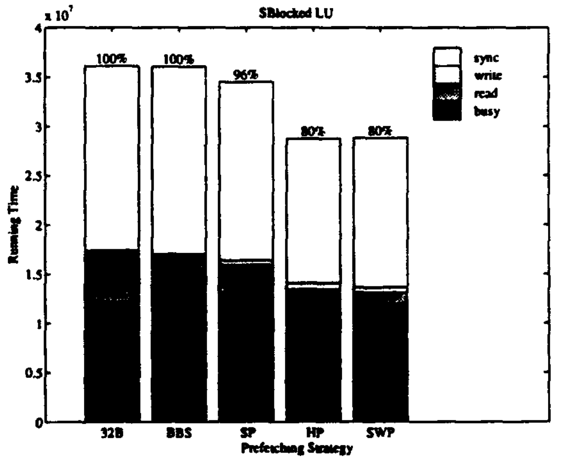

Figure 32: Running time comparison of SBlocked LU under high bandwidth.

5 Conclusions

In this paper, we used execution-driven simulation of parallel programs on a scalable cache-coherent machine to study the performance of three cache-miss-initiated prefetching techniques: large cache blocks, sequential prefetching, and hybrid prefetching. Large cache blocks and sequential prefetching are well-known prefetching strategies. Hybrid prefetching is a novel technique combining hardware and software support for stride-directed prefetching.

Our simulation results showed that large cache blocks rarely provide significant performance improvements; the incremental improvement in the miss rate gained by using larger blocks is simply too small to offset a corresponding increase in the miss penalty. Our results also showed that sequential prefetching improves on the performance of large cache blocks by alleviating false sharing and high miss penalties. A comparison of sequential and hybrid prefetching shows that the latter technique performs at least as well as the former, as it can prefetch with large strides between blocks, while being selective about how aggressively to do so. In fact, given sufficiently high bandwidth and regular memory addressing, hybrid prefetching can perform as well as software prefetching. We conclude that among the cache-miss-initiated prefetching techniques we consider, hybrid prefetching is the only technique that offers significant performance improvements for scalable multiprocessors.

Acknowledgements

We would like to thank Wagner Meira for collecting the trace information necessary for the implementation of hybrid prefetching in our applications. We would also like to thank Leonidas Kontothanassis for coming up with the main modification we made to Mp3d.

References

[Agarwal, 1991] A. Agarwal, "Limits on Interconnection Network Performance," *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, Oct 1991.

[Baer and Chen, 1991] J.-L. Baer and T.-F. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," In *Proceedings of Supercomputing '91*, pages 176–186, 1991.

[BBN, 1989] BBN Advanced Computers Inc., *Inside the TC2000*, 1989.

[Bianchini and LeBlanc, 1994] R. Bianchini and T. J. LeBlanc, "Can High Bandwidth and Latency Justify Large Cache Blocks in Scalable Multiprocessors?," In *Proceedings of the 1994 International Conference on Parallel Processing*, August 1994, Extended version published as TR 486, Department of Computer Science, University of Rochester.

[Callahan *et al.*, 1991] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching," *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[Dackland *et al.*, 1992] K. Dackland, E. Elmroth, B. Kagstrom, and C. Van Loan, "Parallel Block Matrix Factorizations on the Shared-Memory Multiprocessor IBM 3090 VF/600J," *The International Journal of Supercomputer Applications*, 6(1):69–97, Spring 1992.

[Dahlgren *et al.*, 1993] F. Dahlgren, M. Dubois, and Per Stenstrom, "Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors," In *Proceedings of the 1993 International Conference on Parallel Processing*, August 1993.

[Dubnicki, 1993] C. Dubnicki, *The Effects of Block Size on the Performance of Coherent Caches in Shared-Memory Multiprocessors*, PhD thesis, Department of Computer Science, University of Rochester, July 1993.

[Dubois *et al.*, 1993] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrom, "The Detection and Elimination of Useless Misses in Multiprocessors," In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 88–97, May 1993.

[Eggers and Katz, 1989] S. J. Eggers and R. H. Katz, "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs," In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, April 1989.

[Fu and Patel, 1992] J. W. C. Fu and J. H. Patel, "Stride Directed Prefetching in Scalar Processors," In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 102–110, December 1992.

[Lee *et al.*, 1987] R. L. Lee, P.-C. Yew, and D. H. Lawrie, "Multiprocessor Cache Design Considerations," In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 253–262, June 1987.

[Lenoski *et al.*, 1990] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[Mowry *et al.*, 1992] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–75, October 1992.

[Przybylski, 1990] S. Przybylski, "The Performance Impact of Block Sizes and Fetch Strategies," In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 160–169, Seattle, WA, 1990.

[Singh *et al.*, 1992] J.P. Singh, W-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News*, 20(1):5–44, March 1992.

[Smith, 1978] A. J. Smith, "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computer*, 11(12):7–21, December 1978.

[Veenstra, 1993] J. E. Veenstra, "Mint Tutorial and User Manual," Technical Report 452, Department of Computer Science, University of Rochester, July 1993.